3D Gyrokinetic Particle–In–Cell Codes On The TC2000 And CM2 *

Timothy J. Williams and Y. Matsuda

Magnetic Fusion Energy Theory and Computations Group
Lawrence Livermore National Laboratory
Livermore, California 94550

Appeared in
The 1992 MPCI Yearly Report: Harnessing the Killer Micros
UCRL–ID–107022-92
August 1992

Abstract

We have ported a three–dimensional, electrostatic, gyrokinetic, particle–in–cell (PIC) plasma simulation code from the Cray2 to the BBN TC2000 and the Thinking Machines CM2. On the TC2000, we employed the PFP split–join programming paradigm[**?**]; on the CM2 we employed the CM Fortran data–parallel programming language. Though our hopes to produce codes ten or more times faster than the Cray2 code were not realized, we did produce codes which run faster than the Cray2 code on much less expensive hardware—proof of the promise of massively parallel computing. We are now able to run plasma physics simulations with these parallel codes, including some which are not practical on the Cray2.

1 Gyrokinetic PIC

1.1 Simulation Goals

Gyrokinetic particle-in-cell (PIC) simulation[**?**] is an important tool for studying low-frequency ($\omega \ll \omega_{ci}$) plasma instabilities and the associated transport of particles and heat in tokamaks. In spite of the efficiency gained by not following ion cyclotron motion, more efficiency is needed in performing simulations on a transport timescale. We have expended much effort to use additional physics simplifications and algorithm improvements to reduce the computational cost of our vectorized Cray2 simulations. An example of a physics simplifications is the use of the δf method, in which the particles sample the perturbation of the phase–space distribution function away from a fixed background rather than the entire distribution function; this, when coupled with a partial linearization of the equations, can result in greatly reduced simulation noise—allowing the use of fewer particles[**?**][**?**]. Examples of algorithm improvements we employ are electron subcycling[**?**][**?**][**?**] and implicit timestepping schemes[**?**].

We plan to use massively parallel computation to achieve the computational speeds needed to run simulations with sufficient resolution and physics content to believably model turbulent transport in tokamak plasmas. Examples of physics properties missing from our current simulation codes are magnetic fluctuations (finite–β), toroidal geometry effects (highly relevant for the toroidal tokamak plasmas), and particle collisions. All of these effects are being built into Cray2 simulation codes, but all will add computational cost—making it even more essential to use massively parallel machines which allow running larger and longer simulations than the Cray2 allows.

*This work performed by LLNL under DoE contract No. W–7405–ENG–48.

1.2 Simulation Method

The basic ideas behind gyrokinetic PIC simulation are the same as those behind conventional PIC simulation. Particle-in-cell codes simulate plasmas using superparticles moving under self-consistent electromagnetic fields defined on a spatial grid; each superparticle represents many physical plasma particles. The (electromagnetic) force on each particle is computed by interpolating the electromagnetic fields from some set of neighboring points in the field grid. From these forces, accelerations are computed to update velocities, which in turn are applied to update the particle positions. Charge densities (and current densities, for non–electrostatic simulations) are then accumulated from the particle positions (and velocities) onto field grids, using an interpolation formula to compute each particle's contribution to a small set of nearby grid points. The field equations (Maxwell's equations) are then solved on the grid to yield updated electromagnetic fields; this closes the self–consistent loop, allowing computation of new forces to push the particles again, *etc.*

1.3 Computational Characteristics

The interpolation of forces from field-grid arrays to particle positions, and the complementary accumulation of densities from particle data to grid arrays are key elements of the PIC method. Another key element is the solution of the field equations, which are elliptic equations (boundary–value problems). Depending on the parallel programming paradigm used, these different elements present different types of problems for parallel simulation.

One paradigm is domain–decomposition and message–passing (DDMP). One decomposes the spatial simulation domain into subdomains; each processor used computes data on one subdomain. The gathers and scatters for the force computation and charge accumulation needn't be parallelized—each processor does them serially on only the data in its domain, which is stored in that processor's local memory. As the particles move across the subdomain boundaries, and their data is passed to neighboring processors in messages, there is the potential for load imbalance to develop because regions of high and low particle density appear. This load imbalance can be reduced or eliminated by dynamically revising the subdomains. For *gyrokinetic* PIC simulations this is not an extremely important issue, because the mathematical formulation precludes large nonuniformities in particle densities. A problem faced by the DDMP programmer is difficulty in solving the field equations—for which FFT methods are typically used—on the domain–decomposed field–grid arrays. Liewer *et. al.* report that their conventional PIC simulations on purely distributed–memory MIMD machines are not as limited by load imbalance, which they control with dynamic subdomain revision, as they are limited by the expense of the FFT computation for solving the field equations.[**?**][**?**] In contrast, the FFT–based field equation solution is $O(1\%)$ of the run time in our Cray2 simulations.

Another paradigm is the data–parallel model traditionally used in programming the CM2. From the programmer's viewpoint, the data arrays are all treated as globally–addressable (*i.e.*, arbitrary sections of the data arrays are referenced in the code as single objects; the compiler and hardware automatically distribute the data elements among the processors). The gathers and scatters for the force computation and charge accumulation require communication of data between particle and field–grid arrays in a random pattern. These gathers and scatters, even when optimized using simulated annealing methods, have been shown to burn as much as 75% of the computer time in gyrokinetic PIC simulations on the CM2[**?**]. In contrast with DDMP, this method can work well for solution of the field equations—especially if they are solved using the very fast FFT's possible on the CM2, for example.

Yet another paradigm of particular interest to the MPCI is the split–join model embodied in PCP and PFP[**?**]. This paradigm provides explicit support for both local memory and globally–addressable memory; for this reason, it is particularly well–suited to the TC2000. Sticking strictly to the PFP paradigm requires storing at least the field–grid data as globally–addressable arrays in order to solve the boundary–value field equations. This implies the same randomized gathers and scatters as in the data–parallel paradigm, but the extremely high bandwidth and uniform interconnectedness of the TC2000 processor network allows this to happen more efficiently than on the CM2 (this will be detailed in a future publication). This method can also work well for solving the field equations; very fast multidimensional FFT's are possible if one makes use of the local/global memory heirarchy on the TC2000 and designed into the PFP paradigm.

2 TC2000

2.1 The Machine

Refer to our earlier report[**?**] for a description of the TC2000 hardware and a horsepower comparison with the Cray2 and CM2. The key features of the TC2000 for our programming purposes are that (1) it supports hardware global data memory access as well as local memory access, (2) it's readily available to us here at LLNL, and (3) it provides excellent parallel debugging and analysis tools for code development. Our plan for the TC2000 has been to use it as a development platform for parallel codes—to do scaling studies, but not necessarily to do production physics runs.

2.2 PFP

As mentioned in Section **??**, we employ the PCP/PFP parallel programming paradigms on the TC2000. The Parallel C Preprocessor and Parallel Fortran Preprocessor (PCP and PFP)[**?**][**?**] are implementations of the *split-join* parallel programming paradigm which are descended from SPMD model [**?**] and The Force [**?**]. In this paradigm, a fixed-CPU-count *team* of processors enters the code; all members execute the code unless instructed otherwise by special preprocessed control statements embedded in the C or Fortran code. The team has a *master* processor, which can be specifically accessed for scalar code blocks or other special purposes. The team can be *split* to do independent code blocks or to execute the same code on independent data.

To use PCP/PFP, one inserts special kewords and control statements into his plain C/Fortran code. These are preprocessed and the output is fed into the C/Fortran compiler. The number and variety of these extra coding constructs is limited, making the system relatively easy to program with. To run the codes, one specifies the fixed number of processors in the team which enters the code before the execution begins. Our earlier report[**?**] includes some coding examples for PIC codes along with this general outline of the PCP/PFP ideas.

2.3 Data Layout

The particle data consists of position and velocity arrays for each of the three dimensions. The particles are divided equally among all the processors; each stores its equal share of the particle data in its local memory. Each particle array is a floating–point vector of length $N_{particles}/N_{processors}$.

The field data (grid–array data) is stored in a globally–addressable 3D floating–point array, using the shared interleaved storage provided on the TC2000. For this type of electrostatic simulation, there are at most four arrays: one for the scalar potential, one for the charge density, and three for the three components of the electric field. We have in fact eliminated the use of arrays to store the electric field; we compute the electric field for the force computation using analytic derivatives of the interpolation formula.[**?**] For problems which are small enough, it is sometimes faster in absolute execution time for each processor to store a local copy of the entire charge density grid array, accumulate from it's local particle data into that local copy, then do a locked sum into the global array (see Section **??** below).

2.4 Scalings

2.4.1 Force computation and particle push

Given the accelerations for all particles, the timestepping of the velocities and then the positions is perfectly parallel over the number of particles in the simulation. The only source of inefficiency in this part of the program is switch contention in the global memory fetch from the field arrays which must be done for every particle in order to interpolate the electromagnetic forces to the particle's position. Figure **??** shows linear scaling of the force computation and particle push with $N_{processors}$. If we define the parallel efficiency to be

$$\epsilon \equiv \frac{1}{N_{processors}} \times \frac{T_1}{T_{N_{processors}}},$$

where T_1 is the time for execution on one processor, we measure $\epsilon = 0.95$ for 118 processors from the data used to produce Figure **??**. This efficiency could possibly be increased even more by enabling caching of data from the field arrays before reading them in the force–gather loop.

2.4.2 Charge accumulation

As described in our earlier report[**?**], this part of the PIC algorithm is tricky to do in parallel because of the data dependency inherent in the scatter operation. We have investigated three ways to attack this problem:

(1) Maintain local copies of the entire charge density array on all processors. This makes the accumulation operation for each processor's particles a purely local–memory operation, scattering from the local particle data array into the local grid–array. Each local array contains a part of the total charge density; these

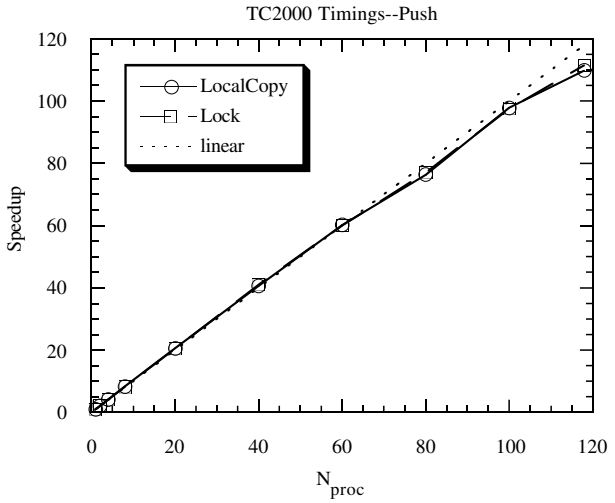

Figure 1: Measured speedups of the particle push, including the force computation (gather) in the TC2000 code. The two different charge accumulation methods described later in the text have no impact on this part of the code. The short–dashed curve is the theoretical ideal linear speedup. Run parameters: 115351 particles, 32768 grid cells.

must all be summed together into the global array to yield that total charge density. Each processor must lock each element of the global array as it adds in its contribution. Rather than use the general lock function provided by PFP, we use a lightweight floating–point atomic add assembler function from BBN; basically, this procedure uses the floating–point number to be added as a semaphore[**?**]. For small–grid systems with many particles per grid cell, and for larger–grid systems run with a small enough number of processors, this method of parallel charge accumulation is the fastest in absolute execution time because it makes the most use of local memory versus global memory. Of course, it is only applicable to problems for which the grid is small enough to store the local copies on each processor. There is a fundamental limit on *scaling* of this method with $N_{processors}$, though: it can never be done faster than the amount of time it takes a single processor to loop through all the grid cells. This is because each processor must loop through all the cells in its local grid array when it sums them into the global array to get the total charge density. The flattening of the solid curve in Figure **??** shows this limit. Lock–conflicts with other processors are another drag on this method, as exhibited by the slight downward turn of this curve at high processor count. The measured parallel efficiency is $\epsilon = 0.08$ with 118 processors.

(2) Use only the global charge density array. As it loops through its particles, each processor must lock the appropriate element of the charge density array for each particle. Even though each particle contributes to a small neighborhood of grid cells, according to an interpolation formula, only one cell at a time must be locked. The sources of inefficiency in this method are the time spend calling the lock function itself and the time spent waiting for a lock to clear in the event that another processor is locking the same element. As in method (1), we use an assembly–language floating–point atomic addition routine which is probably the fastest possible way to implement a lock function on the TC2000. The large–dash curve in Figure **??** shows the scaling of this method with processor count. Clearly the scaling is better than method (1), but the measured parallel efficiency $\epsilon = 0.5$ with 118 processors is still the efficiency bottleneck for this code. The choice whether to use method (1) or method (2) in practice is not as clear as the scaling curves would indicate. The curves in Figure **??** show the absolute execution times of the charge accumulation for the two methods. For small processor counts method (1) is faster (and the entire code is consequently faster); for large processor counts method (2) eventually wins out, and the entire code is also faster with this method.

(3) Sort the particle data. In early versions of the code, the particle data was stored in global arrays rather than divided up into local arrays. As described in our earlier report[**?**], sorting methods such as those used to vectorize charge accumulation on vector supercomputer PIC codes can be employed to parallelize the operation on massively parallel machines. We implemented a parallel quicksort algorithm in a 1D PIC code as a demonstration of the method, but our sort scaled poorly with processor count and was a bottleneck for code execution speed. Since then we have not found a parallel sorting method which scales linearly with processor count and correctly sorts our particle data. With our new local storage of particle data, the situation has changed; we are currently pursuing other ways to improve our parallel PIC codes, but this subject may ultimately be revisited if the nodes in massively parallel machines become vector processors.

2.4.3 FFT

We have modified a multidimensional complex–to–complex parallel FFT algorithm developed by A. Berno[**?**] to perform the real–to–complex FFT's used in our solution of the field equations. The basic method is a time–honored one in parallel computation:

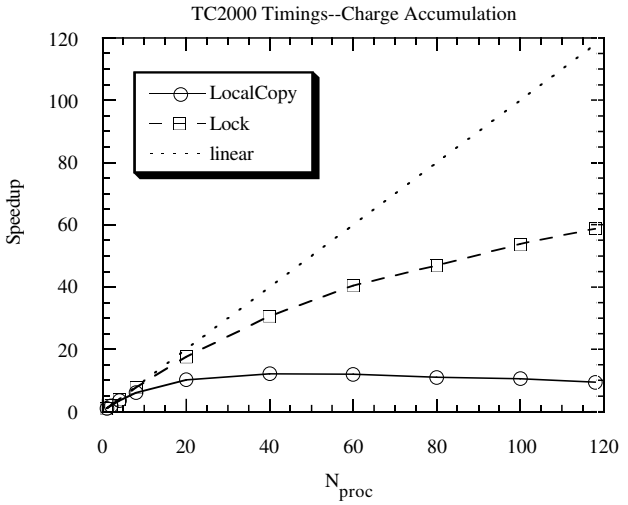

Figure 2: Measured speedups of charge accumulation (gather) in the TC2000 Code, using two different methods described in the text. The short–dashed curve is the theoretical ideal linear speedup. Run parameters: 115351 particles, 32768 grid cells.

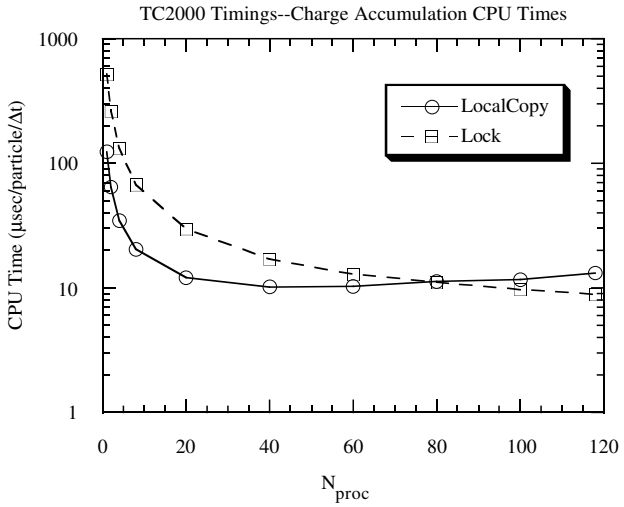

Figure 3: Measured actual CPU times for charge accumulation (gather) in the TC2000 Code, using two different methods described in the text. Run parameters: 115351 particles, 32768 grid cells.

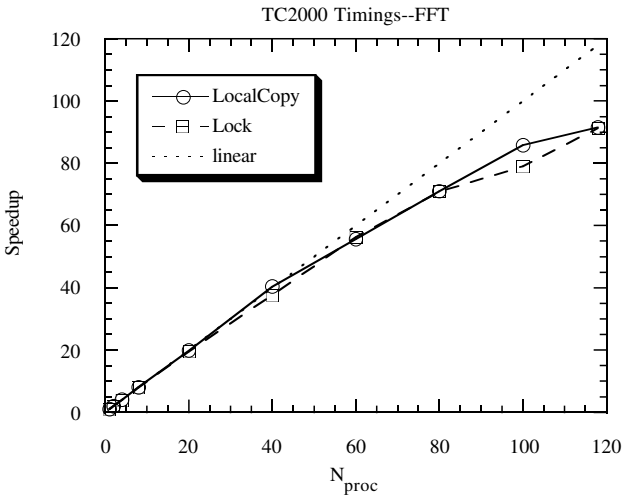

Figure 4: Measured speedups of the parallel FFT in the TC2000 Code. Note that the two different charge accumulation methods have no impact on this part of the code. The short–dashed curve is the theoretical ideal linear speedup. Run parameters: 115351 particles, 32768 grid cells.

perform serial FFT's in one dimension, effecting parallelism along the other two dimensions. The processors transform (in parallel) 1D strips along one of the three dimensions, then synchronize and transform 1D strips along the second dimension, then do the same for the third. The minimum amount of parallel work available goes like half the smallest product of the numbers of grid cells between any two dimensions of the simulation grid. (The factor of a half comes in because this is a real–to–complex FFT, and two 1D strips of data from the array are combined together and treated as a single complex strip for processing.) This FFT method takes advantage of the memory heirarchy on the TC2000 by doing simple copies back and forth from strips of global array elements to local 1D vectors. The 1D transforms are thereby rendered completely local in memory. Figure **??** shows the scaling of this FFT with processor count. The measured parallel efficiency at 118 processors is $\epsilon = 0.77$. This simulation has a $32 \times 32 \times 32$ grid, so the minimum number of 1D strips available for parallel processing is $32 * 32 * 0.5 = 512$; a more precise study of the efficiency of this part of the algorithm will be presented in a future publication.

2.4.4 Field equation solve

We solve the Poisson equation in **k**-space, employing the aforementioned parallel FFT. There is an iteration

involved because of our representation of the electrons in the plasma by a simple Boltzmann response, but basically the solution reduces to an algebraic equation in **k**-space. This reduction alone is a strong motivator for using Fourier methods for field equations in PIC codes, but additional complications peculiar to the *gyrokinetic* field equations make it even more desirable. (Specifically, there is a simple **k**-space representation of a Bessel function which in general is a path integral in configuration space.) Figure **??** shows the scaling of this field–equation solution, *exclusive of the FFT operations*, with processor count. There are large fluctuations in the measurements because this is a small fraction of the total code execution time, and we are seeing the precision limits of our timing method.

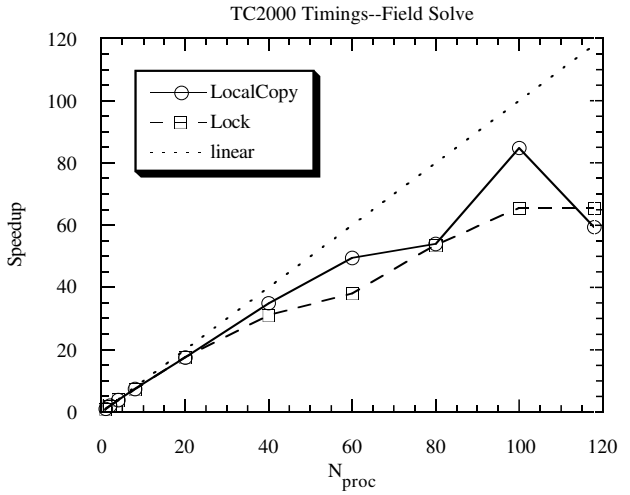

Figure 5: Measured speedups of field equation solve (excluding FFT) in the TC2000 Code. Note that the two different charge accumulation methods have no impact on this part of the code. The fluctuations in the data values are caused by imprecision of our timing method for this small part of the overall code execution time. The short–dashed curve is the theoretical ideal linear speedup. Run parameters: 115351 particles, 32768 grid cells.

2.4.5 Whole code

Figure **??** shows the scaling of the whole PIC code with processor count. The efficiency of the code is dependent on a number of parameters, a key one being the number of particles per cell. More detailed parameter studies will be presented in a future publication. These curves demonstrate the aforementioned claim that the local–grid–copy method is faster at low processor counts and slower at high processor counts. The parallel efficiency at 118 processors is $\epsilon = 0.65$ for the local–grid–copy method. The relative contributions of the different parts of the code to the execution times at 118 processors are summarized in the following table. The "Field Solve" entry excludes the FFT time, which is listed separately. The "Push" entry includes the force computation. The "LocalCopy" and "Lock" column headings refer to methods (1) and (2) of doing the parallel charge accumulation. The total execution times are in microseconds per particle per timestep.

BREAKDOWN OF EXECUTION TIME		
	LocalCopy	Lock
Charge Accumulation	61.4%	52.4%
FFT	14.2%	18.0%
Field Solve	6.3%	7.2%
Push	18.0%	22.4%
Total Execution Time (μsec/particle/Δt)	21.4 μsec	16.9 μsec

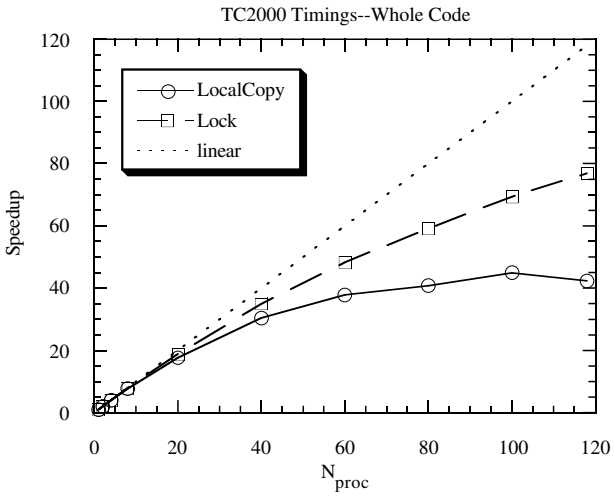

Figure 6: Measured speedups of whole simulation code on the TC2000 Code, using two different charge accumulation methods described in the text. The short–dashed curve is the theoretical ideal linear speedup. Run parameters: 115351 particles, 32768 grid cells.

3 CM2

3.1 The Machine

Refer to our earlier report[**?**] for a description of the CM2 hardware and a horsepower comparison with the Cray2 and TC2000. The key features of the CM2 for our programming purposes are that (1) it provides the CM Fortran data–parallel programming language,

which implements parallelism in a rather straightforward translation from the Cray2 code; (2) the machine at Los Alamos National Laboratory is readily available to us for development work; and (3) it has the potential horsepower to run very fast, and to run very large codes. Our hope has been to use this machine to do new production physics simulations in regimes unreachable with the Cray2.

3.2 CM Fortran

As mentioned in **??**, and in our earlier report[**?**], we employ the CM Fortran data–parallel programming language on the CM2. CM Fortran is a sub/superset of Fortran 90; it uses array syntax rather than looping, and includes statements giving some control over how the data is aligned on the virtual processor mesh in the machine. Our earlier report[**?**] includes some coding examples for PIC codes along with this general outline of the CM Fortran ideas.

3.3 Data Layout

The key data structures in the program—the particle data arrays and the field–grid arrays—are stored in CM2 arrays laid out in the canonical way across the mesh of virtual processors. The particle data consists of 1D position and velocity floating–point arrays for each of the three dimensions. Each particle array is a floating-point vector of length $N_{particles}$. The field data is stored in 3D floating–point arrays. Since the particle arrays and grid arrays are of different shape, and since the locations of the particles in the spatial grid change randomly as the simulation evolves, there is no regular pattern of communication between the elements of the particle arrays and the elements of the grid arrays. The communication happens during the force computation (gather) and charge accumulation (scatter).

The gather involves reading a finite set of grid array elements for each particle to compute the force for that particle. The acceleration associated with that force then updates the particle's velocity, which in turn updates the particle's position. No special means are employed for retrieving the grid array elements; the CM2 router hardware takes care of setting up the communication paths through its internal hypercube connection topology. One would expect this to be formally less efficient than the equivalent read from the global field grid array on the TC2000 because of the varying lengths of the communication paths on the CM2; a single fetch may involve hopping across and back across many nodes, while it involves only a single switch connection–path on the TC2000. Technically, this operation on the CM2 is handled by the `forall` statement of CM Fortran.

The scatter involves adding charge–density contributions to a finite set of grid array elements for each particle. Because many particles may contribute to the same grid cell, there is a data dependency to be resolved. We use the CM2 library function `CMF_SEND_ADD` for this purpose; this is the equivalent of the floating–point atomic add used for the charge accumulation into the global charge density array on the TC2000.

3.4 Scalings

The conventional methods of computing parallel efficiencies and scalings with processor count do not apply on the CM2 because the machine is not a collection of independently–programmable processors. In particular, one can't run on *one* processor. However, a fully–configured CM2

is divided into 4 quadrants of 16K one–bit processors, so it is possible to check for linearity by running the same problem on one quadrant, two quadrants, and the whole machine. Unfortunately, using more than one quadrant of LANL CM2 requires special bureaucratic procedures, so we have not yet gathered sufficient data to measure scaling in many different paramater regimes. Figure **??**, however, is a reasonable example; it shows the overall code execution time for a fixed problem using one quarter, one half, and all of the one–bit processors. This run had 65536 particles and 65536 grid cells. Roughly, the CPU time scales down linearly with processor count.

If we define a modified parallel efficiency

$$\epsilon' \equiv \frac{16\text{K}}{N_{proc}} \times \frac{T_{16\text{K}}}{T_{N_{proc}}},$$

we measure $\epsilon'_{32\text{K}} = 0.95$ and $\epsilon'_{64\text{K}} = 0.79$ As mentioned in our earlier report[**?**], our emphasis on this machine has always been to seek maximum absolute computational speeds without as much regard for efficiency as is the case on the TC2000, where efficiency measurements are easier and where we had expected lower absolute computational speeds.

includegraphics[width=3.25in,trim=100 235 100 205]CM2Timings.eps

Figure 7: Timings of CM2 code using 16K, 32K, and 64K bit–processors. Run parameters: 64K particles, 64K grid cells.

4 Absolute Timings

The following table sumarizes our absolute code timing measurements on the Cray2, TC2000, and CM2 codes. The times are in a conventional unit used for timing PIC codes: microseconds per particle per timestep. The Cray2 code is fully vectorized and scalar optimized by the CRI FPP fortran preprocessor and CFT77 compiler; it is also optimized by the CRI FMP fortran preprocessor, which inserts compler directives for autotasking (loop–level multiprocessing). It was run on the NERSC 8–processor Cray2, under a night–time system load. Because there is no equivalent to the MPCI Gang Scheduler to guarantee run–time on more than one processor, multitasking performance is poor (in fact, the code runs faster on one processor than it does on eight). The CM2 code was run on the whole Los Alamos ACL machine in batch mode (i.e., with no other processes running or resident in the machine). The TC2000 code was run on 118 processors in benchmark (single–user) mode. (*N.B.*: These runs were also made in interactive (multi–user) mode, which yielded almost exactly the same timings; this indicates that butterfly switch contention with other users is not a major factor for running this code.) The highest scalar Fortran optimization level which produced correct answers was used. For Run I, the local–grid–copy method for charge accumulation was used; for Run II, the global–array–only method was used. The "BIT" annotations in the tables refer to the precision of floating point numbers. Runs I and II are for two different parameter sets: Run I has 269001 particles and 16K grid cells; Run II has 1M particles and 256K grid cells.

64–BIT FLOATING–POINT TIMINGS (μsec/particle/Δt)		
	Run I	Run II
Cray2	30μsec	30μsec
TC2000	15μsec	26μsec
CM2	33μsec	26μsec
32–BIT FLOATING–POINT TIMINGS (μsec/particle/Δt)		
	Run I	Run II
TC2000	9μsec	15μsec
CM2	26μsec	20μsec

Basically, these results demonstrate that our codes on the massively parallel machines are capable of exceeding the performance of the optimized Cray2 code. The parallel machines now offer three advantages over running on the Cray2: faster codes, the added computational and memory efficiency of 32-bit floating–point arithmetic, and the larger total available memory on CM2. By making use of these advantages, we are now in a position to run physics simulations which were previously impractical or impossible on the Cray2. The high performance of the TC2000 code is a pleasant surprise. Contrary to our original plans to use the TC2000 only as a scaling–study machine, we are now planning some production physics simulations for this machine as well as for the CM2.

5 CM2 + TC2000 = CM5

Our current research includes work in collaboration with Eric Salo to effect a domain–decomposition of the field grid arrays. This will eliminate the global memory access and data dependency in the charge accumulation, but it will require a dynamic load–balancing scheme to transfer particles among processors as they move across the subdomain boundaries; it will probably pass the data as messages. Furthermore, as mentioned in Section **??**, this might also require some special means to solve the field equations on the domain–decomposed field–grid arrays (whose subdomains reside in different processors' local memories). We hope, however, to work around this difficulty by maintaining global memory access to the field–grid arrays. We plan to attempt to use the forthcoming Parallel Data Distribution Preprocessor (PDDP)[**?**] to block–distribute a *globally–addressable* field grid array across the processors such that each block is, in fact, a subdomain. This would allow continued use of our current parallel FFT algorithm for solving for the field equations.

This code design, we believe, may be a useful prototype for porting the simulation code to the CM5. It may be that the hybrid split–join/DDMP algorithm, when combined with aspects of our data–parallel CM2 code, can yield an efficient match to the hardware and software of the CM5.

6 In Memory of Yoshi Matsuda

The tragic death of Yoshiyuki Matsuda in October of 1991 shocked and saddened those of us who knew and worked with him. Some of the results in this report represent his last work. Yoshi's optimistic vision of the promise of massively parallel computing, and his enthusiasm in pursuing that promise, were an inspiration to me especially. The work presented here and its continuation keep alive in our memories this

one small aspect of a person who is greatly missed. —T. Williams, April 1992.

References

[1] Eugene D. Brooks III, UCRL-99673, LLNL (1988).

[2] W. W. Lee, J. Comput. Phys. **72** 243 (1987)

[3] A. M. Dimits and W. W. Lee, "Partially Linearized Algorithms in Gyrokinetic Particle Simulations,"PPPL–2718, Oct. 1990.

[4] M. Kotchenreuther, *Bull. Am. Phys. Soc.* **34**, 2107 (1988).

[5] T. J. Williams,"Improved Gyrokinetic Simulation Techniques," *Proceedings of the 13th International Conference on the Numerical Simulation of Plasmas*, (Sept. 1989).

[6] B. I. Cohen, *et. al.*,"Gyrokinetic Code Development,"*Bull. Am. Phys. Soc.*, **35**, 2038, (Nov. 1990).

[7] Timothy J. Williams, B. I. Cohen, and R. D. Sydora, "Optimizing Gyrokinetic Simulation Techniques,"*Bull. Am. Phys. Soc.*, **34**, 2044, (Nov. 1989).

[8] B. I. Cohen and Timothy J. Williams, "Semi–Implicit Particle Simulation of Kinetic Plasma Phenomena,"J. Comput. Phys. **97**, 224 (1991).

[9] P. C. Liewer and R. D. Ferraro, "A 2D Electromagnetic PIC Code for Distributed Memory Parallel Computers", *Proceedings of the 14th International Conference on the Numerical Simulation of Plasmas*, (Sept. 1991).

[10] P. C. Liewer and V. K. Decyk, J. Comput. Phys. **85**, 302 (1989).

[11] J. V. W. Reynders and W. W. Lee, "3D Electrostatic Gyrokinetic Simulations on Parallel Architectures,", *Proceedings of the 14th International Conference on the Numerical Simulation of Plasmas*, (Sept. 1991).

[12] T. J. Williams, Y. Matsuda, and E. Boerner,"Parallel Particle Simulation On the TC2000 and CM–2," in *The MPCI Yearly Report: The Attack of the Killer Micros*, UCRL–ID–107022 (1991)

[13] PFP authors: E. D. Brooks III and K. Warren, LLNL.

[14] *e.g.,* see A. H. Karp, "Programming for Parallelism," Computer **20**, 43 (1987).

[15] H. F. Jordan *et.al.*,"The Force: A Highly Porable Parallel Processing Language," *Proc. 1989 International Conference on Parallel Processing*, IEEE **II**, 112 (1989).

[16] A. .M. Dimits and J. Byers, private communication.

[17] D. Foster, B. Gorda, private communication.

[18] A. Berno, private communication.

[19] PDDP authors: Brent Gorda and Karen Warren, LLNL.